Pascal
NEWSLETTER

Number 8 OREGON SOFTWARE Spring/Summer 1984

Resident libraries save memory

by Steve Poulsen

Prior to Version 2.1B, a Pascal-2 resident library contained both Pascal support library subroutines and File Control Services I/O routines from the system library (LB:[1,1]SYSLIB.OLB). You can now build Pascal resident libraries that do not contain File Control Services (FCS) subroutines. You can also create a memory resident overlaid Pascal library which can be clustered with an FCS resident library such as FCSRES. These new options both save substantial amounts of memory.[1]

The use of cluster libraries greatly reduces task sizes and the amount of virtual memory used, increasing memory available for code and data. System swapping overhead is reduced because each task is smaller and more tasks can fit in memory. Disk space is also conserved because each task image does not contain code from the Pascal library and the system library.

To build a Pascal resident library, first install your Pascal system with the PASBLD command file. Be sure that the Pascal support library LB:[1,1]PASLIB.OLB is the current version. Then use the command file PASRES.CMD. It asks you whether you want to build a memory resident overlaid version of the Pascal support library. If your system supports virtual (PLAS) overlays, select this option. The Pascal support library is built as two overlays. The total physical memory required for the library varies between 6K and 8K words, depending on your hardware configuration. The advantage of virtual overlays is that the library is mapped into your task using only one Active Page Register (APR). This means that the 6K to 8K words of library code can be accessed using only 4K words of virtual address space, for a savings of 2K to 4K words.

If you do not wish to use virtual overlays, PASRES.CMD can build a non-overlaid resident library, which requires 4K words of memory. This may be necessary on some RSX systems, such as VAX/VMS in compatibility mode, which do not support virtual overlays.

After PASRES.CMD builds the library and its associated symbol table, you need to create a partition called PASRES and install the resident library in that partition. (You can use the program VMR to make the partition permanent.)

Before you build the partition, you must examine the map (PASRES.MAP) to determine the total size of the library. See the portion of a sample PASRES map that follows for an example of how to do this.

The task size is given (in decimal words) in the summary at the top of the map. Convert this value to octal bytes and round up to the next multiple of 100(8): 6464. words works out to 31200 octal bytes. The last two zeroes are dropped. Base is the base address where the partition will be loaded.

Note that even though the task size is 6464. words, the task address limits are only 160000 through 177777, or 4K words. The address limits are reduced because virtual overlays are being used. This is reflected in the overlay description which shows that the segments PASIO and PASFLT overlay each other. The savings realized by using virtual overlays is 4878 bytes in this case.

Sample PASRES Map

```
Partition name : PASRES
Identification : 2.1B
Task  UIC      : [2,16]
Task attributes: -HD
Total address windows: 1.
Task  image  size  : 6464. words
Task address limits: 160000 177777
R-W disk blk limits: 000003 000034 000032 00026.
```

[1]Refer to the Pascal-2 User's Manual, Version 2.1B for RSX-11.

In this issue...

```
PASRES.TSK;1 Overlay description:

Base    Top       Length
----    ---       ------
160000  157777  000000  00000.     PASROT
160000  177411  017412  07946.          PASIO
160000  171415  011416  04878.          PASFLT
```

Now that you know the task size, you can create the partition, such as shown in Example A, a sample partition allocation for a small RSX system. The first column of numbers after the partition name is the location of the Partition Control Block within the monitor and can be ignored. The second column of numbers is the base address of the partition in memory. The third column is the length of the partition. With partitions, all addresses and sizes are divided by 100 (octal) to reduce the range of the numbers. The size needed for PASRES is 31200 bytes. Use the SET /TOP command to reduce the size of the GEN partition by that amount, as shown in Example B.

Example A: Original Partition Allocation

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

Example B: New Partition Allocation

```
>SET /TOP=GEN:-312
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 464500 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

As you can see, the size of the GEN partition has been reduced by 31200 bytes. Compute the base of the new PASRES partition by adding the GEN partition's base and length (242100 + 464500 = 726600). Now you can create the PASRES partition based at 7266 with a size of 312.

The command SET /MAIN=PASRES:7266:312:COM creates a partition called PASRES, based at 726600 with a size of 31200, shown in Example C. It also marks the partition as a common partition. Using the command INS LB:[1,1]PASRES, you can now install PASRES.TSK into the partition so that it can be used by Pascal programs.

Example C: PASRES Partition

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300SUB  DRIVER -DL:
       066570 152200 001400 SUB DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB (F11ACP)
GEN    066314 242100 424500 MAIN SYS
       042620 242100020000 SUB  (...MCR)
PASRES 043634 726600 031200 MAIN COM
```

To build a Pascal task which uses PASRES, first compile the program in the normal way. Use the LIBR option to specify that PASRES should be used in read-only mode:

```
>TKB
TKB>TEST/FP/CP=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>LIBR=PASRES:RO
TKB>//
```

When you run the program Test, RSX automatically associates the PASRES library with your task. Your program is mapped into low virtual addresses, and PASRES is mapped into the virtual address range from 160000 through 177777.

If your RSX system supports clustered libraries (RSX-11M V4.0 or later), you can use the cluster option to specify that PASRES should be clustered with the FCS resident library FCSRES. The advantage is that the same APR can map both the overlaid PASRES as well as an overlaid FCSRES. In the partition list shown in Example C, FCSRES and PASRES are both about 6K words long, for a total of 12K

words. By clustering the two libraries together, 12K words of code can be mapped with one 4K word APR for a savings of up to 8K words in the program.

PASRES may be clustered with other libraries besides FCSRES, with the restriction that PASRES must be specified first in the CLSTR option.

When you link a Pascal task with FCSRES, you may get an error status of -39 when you try to open a file, indicating that no buffer space is available for the file. When FCSRES is not used, Pascal programs can intercept the FCS entry points which allocate and deallocate file buffers when a file is opened and closed. Pascal converts these calls to allocate memory from the Pascal heap when files are opened. When FCSRES is used, Pascal cannot intercept the FCS calls which allocate file buffers. In this case, FCS will look in the file storage region for space to allocate file buffers. The size of the file storage region should be very small since Pascal usually uses the heap for file buffers.

To avoid such errors when using FCSRES, allocate space in the program section $$FSR1 for buffers for all the files you intend to open at any one time. Each file requires about 528 (decimal) bytes.

You can allocate this space in either of two ways:

1) If you are not using PASRES but are using FCSRES, you can allocate the space in $$FSR1 by using the EXTSCT option.

2) When you are using PASRES, Pascal automatically enables the ACTFIL option of the Task Builder. The ACTFIL option permits you to specify the number of files you will have open at one time. If the value you specify with ACTFIL is greater than one, you also must use the UNITS option to make more logical unit numbers available to the task. The default value for ACTFIL is four open files.

The simple program Test demonstrates three ways to reduce memory requirements. When this program is linked in the standard way, the task size is 9376. words.

```
program Test;
  var
    f: text;
  begin
    rewrite(f, 'TI:');
    writeln(f, 'It works');
  end.
```

This file was compiled with the /NOWALKBACK switch to produce a small object module.

```
>PAS TEST/NOWALKBAC
>TKB TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
```

This program can also be linked with PASRES, reducing the size of the task to 6720. words because most of the code loaded from the Pascal support library is being shared with other tasks in PASRES. The ACTFIL option is used to set the number of open files to one.

```
>TKB
TKB>TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>ACTFIL=1
TKB>LIBR=PASRES:RO
TKB>//
```

You can save more memory by clustering PASRES with FCSRES. In this case, the size of the task is 3296. words, a savings of about 6K words over the size of the original task. When the program Test was linked without PASRES or FCSRES, the task image required 39 disk blocks. When the task is built with a clustered PASRES and FCSRES, the task image takes up only 15 blocks.

```
>TKB
TKB>TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>ACTFIL=1
TKB>CLSTR=PASRES,FCSRES:RO
TKB>//
```

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Syntax checker for Pascal–2 sources checks the lexical and syntactical structure and provides a method for producing paginated listings (see article in this issue). Available from Unit-C Limited, Dominion Way West, Broadwater, Worthing, Sussex, England.

Pascal programmer wanted to design and program a data base system for a network of microcomputers. Contact: John Foley, Carghill Terminal 4, Portland OR 97203, (503) 286-1842.

Syntax checker avoids compilation bottlenecks, speeds software development effort

by Ralph Hodgson

Mr. Ralph Hodgson has worked for Eurotherm since 1976 and is now technical manager for Unit-C Limited, a subsidiary. He is a member of ACM and IEEE and is interested in real-time computing and program development tools.

Whether we are communicating with people or computers, syntax is what gives precise meaning to our words. We learn from an early age that we can be imprecise in our syntax at times, yet still be understood. The people with whom we communicate are able to fill our syntactic gaps through the context of conversation. Later we learn that a computer is not as forgiving and that with a machine, our syntax must be correct and unambiguous. Because of our previous experiences in communicating, most of us find it frustrating to be stymied by the lack of a simple semicolon or end; statement. Our frustration increases as we waste minutes of compilation time before being told that we have not been understood.

Unit-C's Syntax Checker, UPS, is a fast, efficient precompilation tool designed to reduce the frustration of discovering syntax errors. UPS quickly checks Pascal–2 source code for textual and syntactic correctness and produces a listing file without first compiling the source program.

UPS Features

UPS supports all syntax of Pascal–2's implementation, including embedded switches. Vertically arranged, numbered pointers indicate errors appearing in the source code, and all error reporting occurs in the vicinity of the error, as shown in Figure 1.

As a listing utility, UPS reports structural information for each source line in a program. Figure 2 shows the listing that UPS produces including line numbers, nesting,

Figure 1

```
  3  2          PROCEDURE NextDigit(N: integer);
  4  2            VAR J: integer;
  5  2            BEGIN
  6  2  1  1        WHILE N > Base DO
  7  2  1  1          BEGIN
  8  2  2  3          J:=N DIV Base NextDigit(J); N:=N - (J  Base))
                                   0                              0
                                   4                              5
                                   1                              3
                                   0                              2
                                   v                              v
(410) Expression incopmlete, missing operator or semicolon
(532) Unexpected ')' -- Check for matching parenthesis
  9  2  1  3                 END;
 10  2  1  4             CASE Base OF
 11  2  1  5                 1,2,3,4,5,6,7,8,9,10: EmitCh(Chr(N + Ord('0'));
                                                                          0
                                                                          2
                                                                          1
                                                                          3
                                                                          v

(213) ')' expected
 12  2  1  5             OTHERWISE IF N > 10 THEN EmitCh(Chr(N - 10 + Ord('A'))
 13  2  1  8                       ELSE EmitCh(Chr(N + Ord('0')))
 14  2  1  8             END Case
 15  2                 END NextDigit ;
```

Figure 2

```
Pascal Syntax Checker RSX11 V1.1 Site 1312-0  19-Mar-1984 21:57:10 Page 1
Unit-C Ltd Dominion Way West, Broadwater Worthing W. Sussex  BN14 8NT ENGLAND
Copyright 1984 Unit-C Ltd.
listex/list

 1                -- Demonstrates listing features of the syntax checker UPS
 2
 3               PROGRAM ListExample;
 4
 5  1               PROCEDURE ConvertNumber(Number, Base: integer;
 6  1                                          PROCEDURE EmitCh (Ch: Char));
 7  1
 8  2                 PROCEDURE NextDigit(N: integer);
 9  2                    VAR
10  2                       J: integer;
11  2                    BEGIN
12  2  1  1                 WHILE N > Base DO
13  2  1  1                    BEGIN
14  2  2  2                    J := N DIV Base;
15  2  2  3                    NextDigit(J);
16  2  2  4                    N := N - J  Base
17  2  1  4                    END;
18  2  1  5                 CASE Base OF
19  2  1  5                    1, 2, 3, 4, 5, 6, 7, 8, 9, 10:
20  2  1  6                      EmitCh(Chr(N + Ord('0')))
21  2  1  6                    OTHERWISE
22  2  1  8                      IF N > 10 THEN EmitCh(Chr(N - 10 + Ord('A')))
23  2  1  9                      ELSE EmitCh(Chr(N + Ord('0')))
24  2  1  9                    END Case
25  2                    END NextDigit ;
26  1
27  1                 BEGIN ConvertNumber
28  1  1  1              NextDigit(Number)
29  1                 END ConvertNumber;
30  1
31  1               PROCEDURE OutCh(Ch: Char);
32  1                 BEGIN
33  1  1                 BEGIN Nesting
34  1  2                  BEGIN
35  1  3  1                 write(Ch)
36  1  2  1                END
37  1  1  1              END
38  1                 END OutCh ;
39  1
40                  BEGIN ListExample
41     1  2           ConvertNumber(201,  2, OutCh); Writeln;
42     1  4           ConvertNumber(201,  8, OutCh(; Writeln;
43     1  6           ConvertNumber(201, 10, OutCh); Writeln;
44     1  8           ConvertNumber(201, 16, OutCh); Writeln
45                  END.
```

blocking and inclusion levels and statement numbers. You can use this information for checking control flow, statement blocks and lexical nesting procedures. The nesting and blocking levels are particularly useful for programmers faced with unravelling deeply nested program structures.

Running the Syntax Checker

UPS is scheduled in a manner consistent with the running of the Pascal-2 compiler [Editor's note: to "schedule" means to "invoke"]. Some examples are shown in Figure 3. Rather than a schedule command, UPS reports a summary of the manual in the form of help text. This includes a list of recognized options as given in Figure 4. You need to specify options only to lengths sufficient to make their interpretation by UPS unambiguous.

You can easily control the format of a UPS listing. In addition to the directives, **length** and **width**, for page size, controls can process included text and pagination at syntactic points in the source. You can alter the second line of the list banner to provide a unique identification to the listings.

Automatic pagination improves a Pascal listing by inserting page breaks where space is insufficient for a clean start to a new procedure. The directive clause, **break=*n***, causes page breaks at new procedures when there are less than *n* lines remaining for procedural declarations on the current page.

By default, included source text is ignored because a syntax check is usually of no benefit. However, if you require a listing with the included files, give the directive clause **noignore include** as a switch on the schedule line.

To include listings of other sources, for example, command files, UPS has a **nosyntax** option. When you use the **nosyntax** switch, the listing should appear last on the schedule line.

Configuration

A customized message file, UPSA.MSG, configures UPS. The message file can be a shared file for all users or a private file within a specific directory. You configure UPS with a message editor called MED. Editing is allowed on the list banner and the default settings for program options **break**, **length** and **width**. Editing is also allowed on the configuration of UPS to recognize switches of more than

Figure 3

```
1)  To check for syntax errors:-

        UPS myprogram

2)  To obtain a listing:-

        UPS myprogram/list

3)  To obtain listing with page breaks
    at new procedure definitions and bodies:-

        UPS myprogram/list/break=20

4)  To change the width and length
    of the listing:-

        UPS myprogram/list/length=80/width=100

5)  To obtain a listing with included files:-

        UPS myprogram/list/noignore include
```

Figure 4

break=n	Sets the automatic break on procedures, functions and their respective 'bodies' to 'n'.
errors=n	Sets the error tolerance to 'n'. The syntax checker will exit after 'n' errors.
length=n	Sets the length of the list page to 'n'
list	Request a listing to a file or device
ignore options	Ignore specified options(s). A list of directives can be specified separated by commas.
nolist	No listing
noignore option	Re-enable options(s)
nosyntax	Turn off syntax checking. This obtains a listing with no structural information.
width=n	Sets the width of the list page

one implementation of Pascal–2, for example, native and cross products on the same host.

Changes to the list banner may be useful when software must be documented for third party transfer, or when project identity needs to be evident on each listing. MED currently functions only on terminals that support ANSI escape sequences, such as the VT100 family.

You can check the configuration of UPS with a reporter called UCR, which lists the current configuration on the users terminal. Figure 5 illustrates a typical configuration listing.

Figure 5

```
UPS Configuration RSX V1.1FT Site f1312-0
Unit-C Ltd., Dominion Way West, Worthing,
  West Sussex, England

Support From:  Unit-C
Licence Date:  10-Jan-84
Expiry  Date:  10-Jan-84
Recognises Host   Switches For:
  RSX, RT11, RSTS, UNIX
Recognises Target Switches For:
  DEC, 68000
Defaults:-
  Lines Per Page  = 60
  Width of  Page  = 79
  Error Tolerance = 100
  Page Break Trip = 0
```

Product Status

UPS is implemented on RT11 and RSX (see Information Exchange). Work is in progress on a UNIX syntax checker and plans are being made for a VERSAdos version.

Pascal has a standard II

Last summer, the International Standards Organization approved the draft Pascal standard as the new official standard. But we heard that some of the "yes" votes were conditional, and it wasn't clear whether we had an official standard or not. We have confirmed (again) that yes, there is a Pascal standard.

The language is identified as International Standards Organization ISO 7185 and was published in late 1983. The ISO standard is identical to the British standard, and orders should reference the British number, BS 6192. The standard is available from ANSI for $64.00, prepaid. Request document BS 6192 and write: ANSI, International Department, 1430 Broadway, New York, NY 10018.

The ISO standard allows two levels of conformance, Level 1 (including conformant array parameters) and Level 0 (not including conformant array parameters). The American ANSI-IEEE standard is identical to Level 0 of the ISO standard.

Pascal–2 for PDP-11/UNIX conforms to Level 0. All other Pascal–2 native compilers, including 68000/UNIX, conform to Level 1.

NAMRTS utility added for RSTS

Oregon Software still includes the NAMRTS utility with the Pascal–1 and Pascal–2 compilers for RSTS. With this utility, users who are running Pascal–1 and Pascal–2 on their systems at the same time can easily identify which run-time system is associated with which Pascal program. They can then make new associations where necessary.

Because of changes DEC has made in the RSTS operating system, this utility does not work under RSTS version 8.0 or later.

For RSX Systems

Accessing I/O devices from Pascal

by Steve Poulsen

PDP-11 computers control I/O devices by device control registers, located in a special area of memory called the I/O Page. The I/O Page is in the highest 8 KB which can be addressed on the particular PDP-11. The I/O Page on a PDP-11/45 exists in addresses 760000 through 777777.

The RSX operating system usually maintains complete control of all I/O devices. The RSX executive protects the I/O Page from users' programs by means of the memory management hardware.

Each device is controlled by "device control registers", a set of one or more words located in the I/O Page. Programs can access these device control registers much the same as any other words in memory, except that some device control registers are read-only or write-only. Commands are sent to a device by storing a value in a device control register, and the device returns status information and data in the registers. Disks and other high-speed devices return data directly into memory buffers.

)evice drivers can access the I/O Page but writing device .rivers is a complex task and they generally cannot be vritten in Pascal.

'rivileged tasks on RSX also can access the I/O page. Vhen a privileged task is loaded, the RSX executive maps he upper 8 KB of the task's virtual address space to the /O Page. These privileged programs can be written in 'ascal, but this is not recommended because privileged asks also map over the executive. This limits the size of he task, and programs with errors can easily crash the ystem. Privileged programs have other side effects which 'ou may not desire.

[he best way to access the I/O Page from a Pascal program s to use a "device common." A device common is similar o a shared common area. A shared common area provides ı map from a task's virtual addresses to addresses in a ıamed partition in physical memory, while a device common provides a map from a task's virtual addresses to addresses in the I/O page. The device common can provide ıccess to the entire I/O Page, or it can limit access to a 'ange of as little 64 bytes. This limited access helps prevent ɔrograms with errors from causing harm to the system.

WARNING

> Extreme care should be exercised when accessing the I/O Page. If the device being accessed is "known" to the RSX system, interrupts for that device should be temporarily disabled to prevent the device driver from receiving unexpected interrupts.

[he following example, executed on a PDP-11/45, demonstrates how to access the KW11-P programmable clock from a Pascal program. The three clock control registers are located at addresses 772540, 772542, and 772544.[1] The program creates a device common which provides access to addresses 772500 to 772577 in the I/O Page. This range, which contains the clock device registers, is the smallest range which can be mapped with a device common.

As in the case of a shared common area, a dummy file creates the Task Builder data structures required to access the device common. You are mapping 100 bytes, so a simple MACRO file that allocates 100 bytes can be used.

```
.title PCLOCK Programmable clock
                device common
.BLKB 100
.end
```

You cannot use this file to initialize the I/O device itself. Its purpose is to create a Task Builder symbol table file. The macro assembler produces the object module PCLOCK.OBJ from the file PCLOCK.MAC.

```
>MAC PCLOCK=PCLOCK
```

The Task Builder is then used to produce the symbol table file.

```
>TKB
TKB>PCLOCK/-HD,PCLOCK,PCLOCK=PCLOCK
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=PCLOCK:160000:100
TKB>//
```

The /-HD option specifies that no header should be created for the task. The STACK=0 option prevents space from being allocated for a stack. The PAR=PCLOCK:160000:100 option specifies use of the device common area called PCLOCK. 160000 is the virtual address at which the device common appears in each task which maps to the device common. As with a shared common area, the base address must be a multiple of 8K bytes. 160000 was used as the virtual address because it is the highest 8K bytes available in a 64 KB virtual address space. High virtual memory should be used because Pascal tasks extend from low virtual memory to higher virtual memory as more heap space is required. The size of the device common area is the minimum size, 100 bytes.

The creation of a device common is similar to the creation of a shared common area. This command:

```
>SET /MAIN=PCLOCK:7725:1:DEV
```

creates a device common called PCLOCK in the address range of 772500 to 772577 in the I/O Page. Note that addresses and lengths are expressed as multiples of 100 bytes. The PAR command allows you to see the device common:

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       041474 242100 020000 SUB  (...MCR)
PCLOCK 041554 772500 000100 MAIN DEV
```

[1] A complete description of the KW11-P clock can be found in the "PDP-11 Peripherals Handbook."

The device common is made available when PCLOCK.EXE is installed with the INS command:

```
>INS PCLOCK
```

The INS command does not copy PCLOCK.EXE into the device common area. It only makes the device common known to the system and makes it possible for the Task Builder to link tasks to the common area.

When a Pascal task maps to the device common, virtual addresses 160000 to 160077 in the Pascal task are mapped to addresses 772500 to 772577 in the I/O Page. As a result, the clock registers, which start at 772540 in the I/O Page, are located at virtual address 160040 in the Pascal task. In other words, the clock registers at 772540 are offset by 40 bytes from the base of the device common at 772500. Therefore, in the Pascal task, the clock registers are located 40 bytes past the virtual base of the device common (160000) in the Pascal task.

The following program initializes and starts the programmable clock at a 100 kHz rate. It then samples the clock twice to determine how much time was required to compute a natural logarithm.

```
Program Clock;

{  Example showing how to access I/O devices
   from Pascal }

  const
    clock_address = 160040B; { Clock virtual
        address }
  type
    kw11p_csr =
      PACKED RECORD
      { KW11-P Control/Status Register }
        run: boolean; { Start counting }
        rate: (khz_100, line_freq, khz_10,
              external_freq);
        mode: (single_interrupt, repeat_interrupt);
        direction: (down, up);
        fix: boolean;
           { For maintenance operation }
        interrupt_enable: boolean;
        done: boolean;
           { Done counting }
        unused: 0..127;
           { Unused bits in CSR }
        error: boolean;
           { Error detected }
      end;

    clock_registers =
      RECORD
        csr: kw11p_csr;
          { Control/Status register }
        count_set_buffer: integer;
        counter: integer;
      end;

    clock_pointer = ^clock_registers;

  var
    clock: clock_pointer;
     { Pointer to clock control registers }
    start, stop: integer;
     { Start and stop counter values }
    r: real;

  begin
    clock := loophole(clock_pointer, clock_address);
    WITH clock^.csr DO
      BEGIN { Initialize the clock }
      rate := khz_100;
        { 100 kHz clock rate }
      mode := single_interrupt;
      direction := up;
      fix := false;
      interrupt_enable := false;
      end;
    clock^.csr.run := true;
      { Start the clock }
    start := clock^.counter;
    r := ln(1.23456789);
    stop := clock^.counter;

    writeln('Time: ', (stop - start) *
        10: 1, ' micro-seconds');
  end.
```

The record KW11P_CSR defines the clock control and status register which maps the fields in the control register. (See the "PDP-11 Peripherals Handbook" for a definition of the control register.)

The type CLOCK_REGISTERS defines the three clock device registers. The type CLOCK_POINTER is defined as a pointer to CLOCK_REGISTERS, and the variable CLOCK is defined as a CLOCK_POINTER. The variable CLOCK does not allocate the clock registers; it is simply a one-word address.

The connection between the Pascal program and the I/O Page is established when the LOOPHOLE function in the first statement in the program assigns the address of the clock registers (the constant CLOCK_ADDRESS) to the pointer CLOCK. The pointer CLOCK is assigned a virtual address

of 160040. When the program is linked with the device common, virtual address 160040 in the task is mapped to physical address 772540 in the I/O Page. When the program is running, references through the pointer CLOCK access fields in the clock's device registers.

Compile the program with the Pascal–2 compiler. (Pascal–1 users must modify the program to change the packed record and use a variant record to assign a value to a pointer.) When the clock task is built, the device common is treated the same as any other shared common area.

```
>PAS CLOCK
>TKB
TKB>CLOCK/FP/CP,CLOCK=CLOCK,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESCOM=PCLOCK/RW
TKB>//
```

The RESCOM option (above above) causes the Task Builder to read PCLOCK.STB to obtain information needed to map the PCLOCK device common . If PCLOCK.STB and PCLOCK.EXE are copied into LB:[1,1], then instead of the RESCOM option, the COMMON=PCLOCK:RW option could be used.

When the program is run, you see the time required to compute a natural logarithm.

```
>RUN CLOCK
Time: 260 micro-seconds
>RUN CLOCK
Time: 250 micro-seconds
>RUN CLOCK
Time: 250 micro-seconds
```

With the clock counting at 100 kHz, each "tick" represents 10 micro-seconds. This means that the total time is computed by counting the number of ticks and multiplying by 10.

Although this example may seem simple, it has a side effect on the RSX system on which it runs. Pascal programs cannot easily handle device interrupts because the interrupts are processed in kernel mode by highly specialized code. In this example the clock interrupt enable bit is turned off so that you have stopped time on the system:

```
>TIME
22:47:37 09-FEB-84
>TIME
22:47:37 09-FEB-84
>TIME
22:47:37 09-FEB-84
```

RSX uses the clock to keep track of the time of the day. When the clock interrupt is disabled, RSX loses track of the clock. To fix the problem, use the OPEN command to alter the clock control and status register. The following example shows how to re-enable clock interrupts at the proper rate in the proper mode:

```
>OPE 772540
772540 /000220 115<ESC>
```

The value 115 was computed from the information about the clock in the "PDP-11 Peripherals Handbook." It is easier to examine the clock status register to determine the correct control register settings before running the Clock program.

```
>TIME
22:47:49 09-FEB-84
>TIME
22:47:53 09-FEB-84
>TIME
22:48:00 09-FEB-84
```

The clock now seems to be in order, but it may be a good idea to re-boot the system to be sure nothing else has been damaged.

As this example shows, extreme care must be taken to temporarily disable interrupts to the device being accessed, if the device is "known" to the RSX system.

Do not be tempted to try to perform special terminal I/O by directly addressing the terminal's device control registers. With the wide range of control provided by the SET command and the subfunction bits available with the QIO system directive, direct access to terminals should seldom be necessary. The best policy is to only access devices which are not part of the RSX configuration.

I/O switches enhance your terminal's performance, without penalties

by Steve Poulsen

The RSX terminal driver provides several automatic features which may not suit special I/O applications. Using combinations of I/O control switches added in Pascal–2 Version 2.1A, you can control these automatic features without the performance penalties of previous solutions.

During normal operation, the terminal driver itself supplies the line formatting characters: it first prints a line-feed character to skip to a new output line, writes the data in the line to the terminal, then adds a carriage-return character. The terminal driver also automatically inserts carriage-return/line-feed characters when an output line "wraps" around the right edge of the screen or paper. It provides editing functions which delete input characters or entire lines and expand tab characters to spaces.

For some applications, you may prefer to use direct cursor addressing to place output at specific locations on a terminal's display screen. The buffering performed by Pascal and the terminal driver may prevent the output from being printed when you want it to be. Since the terminal driver inserts a line-feed at the start of the line and a carriage-return at the end of the line, it may not be possible to write data to the last line on the screen, for example. The line-feed will cause the entire screen to scroll up.

You may have tried any of three methods to alter terminal driver activity. Each has its drawbacks. The RSX SET command controls many of the terminal driver features. However, these changes are permanent until a subsequent SET command is issued. As explained in the "Pascal–2 V2.1/RSX User Manual," you can specify the /FTN switch and then use the FORTRAN carriage control conventions. But the /FTN switch does not cause data to be written to the terminal immediately. The characters are still buffered until a writeln is executed, causing annoying delays in terminal print out. A third alternative, also in the user manual, is controlling the buffering of input and output with the /BUFF:nnnn switch. System performance suffers greatly with the /BUFF feature because the program must communicate with the terminal driver for each character read or written. In addition, the input editing features are disabled because the terminal driver is not buffering the input.

The following combinations of Pascal–2 I/O switches often obtain all of the control required without the performance penalties of previous three methods.

/RAL/BUFF:1 and /WAL/BUFF:1 Switches

The /RAL (read all bits) switch prevents the terminal drive from interpreting any input characters, such as the delet character (which normally deletes the previous input char acter) and control-U (which normally deletes the entir line). All characters typed are passed to the program. Th /WAL switch (write all bits) disables the automatic lin wrap feature. The combination of /RAL/BUFF:1 on inpu and /WAL/BUFF:1 on output gives you complete control o all terminal I/O. The terminal driver passes all character typed by the user to the program and each character in : write statement is sent immediately to the terminal wit no special interpretation.

/NOCR Switch

With the /NOCR switch, you can disable the automati insertion of carriage control characters by the termina When you use this switch, a writeln statement write the line without any additional carriage control characters You must insert carriage control characters with the CHR(function. This switch is most often used with the /WA switch to prevent the automatic line wrap feature. Us the combination /NOCR/WAL when performing direct curso addressing on a video terminal.

/NOECHO and /RNE Switches

The terminal driver normally echoes input typed by th user. You can disable this feature with the /NOECHO o /RNE (read with no echo) switches. These switches mus be applied to the input file as shown in program RNE:

```
Program RNE;
    var
      pwd: PACKED ARRAY [1..10] OF char;
    procedure Getpassword;
      var
        inp: text;
      begin
        write('Password: ');
        break(output);
        reset(inp, 'TI:/noecho');
        readln(inp, pwd);
        close(inp);
      end;
    begin
      Getpassword;
    end.
```

The BREAK() procedure forces the output of the partial prompt line. You must use it when a partial line is to be printed and either the input file is not the standard input file or the output file is not the standard output file. Normally this procedure is not required because an implicit BREAK(OUTPUT) is performed when input is read from the standard input file.

/RST Switch

Used with an input file, the /RST switch terminates the input line when any non-printing character (any control character) other than a space is typed. You can identify this termination character by examining the internal file structures used by the Pascal support library. The program RST shows how to use the /RST switch. The include file LIBDEF.PAS is used to define the internal support library data structures.

```
Program RST;

  %include 'libdef';

  var
    line: PACKED ARRAY [1..80] OF char;

  function Terminator(VAR f: text): char;

    var
      x: user_file_variable;

    begin { Terminator }
      x := loophole(user_file_variable, f);
      terminator := chr(x^.iosb DIV 256);
    end; { Terminator }


  begin { Main }
    reset(input, 'TI:/RST');
    write('Type a line: ');
    readln(line);
    writeln('ORD(terminator) = ',
       ord(terminator(input)): 1);
  end. { Main }
```

You can use the terminator function defined above with any input file to determine which character terminated the input line. The terminator is always present in the I/O Status Block field, even if the /RST switch is not used.

/CURSOR Switch

The /CURSOR switch enables terminal-independent cursor control. When you open a terminal as a file with the /CURSOR switch, the first two characters on each line are used to specify a column and line number at which the text is to be printed. The terminal driver generates the necessary cursor control sequences to position the cursor. This means that a Pascal program can perform cursor positioning and graphics on a variety of terminals without having to be recoded for each different terminal. (Terminal-independent cursor control is a SYSGEN option, so it may not be present on all RSX systems.)

The first character on each line is the column number (horizontal position) with column 1 being the first column at the left. The second character is the line number, with line 1 being at the top of the screen. If 128 is added to the line number, the screen will be erased before the line is displayed. The program Spiral shows how the /CURSOR switch can be used.

```
Program Spiral;

{ Demonstration of terminal independent
 cursor control }

  const
    increment = 0.1;
    decay = 0.99;

  var
    angle, radius: real;
    x, y: integer;
    f: text;


  begin
    rewrite(f, 'TI:/CURSOR');
    writeln(f, chr(1), chr(129));
    radius := 12.0;
    angle := 0.0;
    WHILE radius > 1.5 DO
      begin
      x := round(2.0 * radius *
           sin(angle)) + 24;
      y := round(radius * cos(angle)) + 12;
      writeln(f, chr(x), chr(y), '*');
      angle := angle + increment;
      radius := radius * decay;
      end;
  end.
```

/RCU, /WBT, and /PR:0 Switches

In the previous example, the /RCU switch restores the cursor position. When you use this switch, the terminal driver saves the current coordinates of the cursor, prints the line,

and then restores the cursor to its original position. This switch is most often used with the /CURSOR switch to update a field on a CRT without disturbing input or output in progress elsewhere on the screen.

The program Timer shows how the /RCU and several other switches can be used. The program detaches from the terminal and then, every five seconds, updates the time of day in the upper right corner of the screen. The /CURSOR switch specifies the location of the time, independent of the terminal being used. The /RCU switch causes the terminal driver to restore the original cursor position after updating the time. (The /CCO switch disables control-O mode before the time is updated and is not essential to this example.)

```
Program Timer;

{  Print time display on terminal }
  var
    mag, unit, h, m, s: integer;
    t: real;

  procedure Wait(VAR magnitude, unit: integer);
    NONPASCAL; { Defined in system library }

  procedure Detach;
    EXTERNAL; { Defined in Pascal library }

  begin
    detach;
    mag := 5;
    unit := 2; { seconds }
    rewrite(output, 'ti:/cco/rcu/cursor/wbt');
    writeln(chr(1), chr(129));
    repeat
      write(chr(66), chr(1));
      t := time;
      h := trunc(t);
      t := (t - h) * 60.0;
      m := trunc(t);
      t := (t - m) * 60.0;
      s := trunc(t);
      write('  ', h: 2, ':');
      IF m < 10 THEN write('0');
      write(m: 1, ':');
      IF s < 10 THEN write('0');
      writeln(s: 1, ' ');
      wait(mag, unit);
    UNTIL false;
  end.
```

The /WBT switch selects "break-through write" mode. This displays output regardless of the current terminal state. If this switch were not used, the time would not be displayed when a program attached to the terminal. The /WBT switch is not necessary, and you may not want to use it because of possible side effects with screen-oriented editors. The /WBT switch requires that the task be privileged. The /PR:0 switch in the Task Build line makes the Pascal program privileged (without overmapping the monitor, which is unnecessary in this case).

To build and run program Timer, use the following commands:

```
>PAS TIMER/NOWALKBACK
>TKB TIMER/FP/CP/PR:0=TIMER,LB:[1,1]PASLIB/LB
>INS TIMER/TASK=...TMR
>TMR
```

At this point, the current time should appear in the upper right hand portion of your screen. If you do not wish to use break-through mode, you can remove the /WBT switch and the /PR:0 switch from the Task Builder command line. Every five seconds, the time is updated. To stop the time display, use: >ABO TMR.

Oregon Software expands...

In the last few months, we have added several new people. We'd like you to know who they are and what they do.

Joel Clark joined Oregon Software in May as a programmer. He got started with computers using a DEC Rainbow for accounting, bidding, and word processing at his own construction company, then studied at Computer Careers Institute and Portland Community College. Joel says he is enjoying learning the UNIX system and the 68000 series. His other interests include hiking and sailing.

Sales representative **Penny Green** brings a varied background to her new position: she wrote the sports column for a local newspaper and has sold real estate and advertising. Penny has a bachelor's degree in mathematics from Portland State University and is now in the M.B.A. program. She enjoys camping and reads history and theology.

Lisa Payne is on the front line as our new receptionist. Lisa previously headed the publicity department of a manufacturing firm and has a strong background in graphics. She started in February this year and says she appreciates the congenial atmosphere of her new position.

Steve Hampson, Senior Software Engineer, earned his M.S. in computer science at Ohio State University. Before

coming to Oregon Software, he spent five years with the language development group of an Ohio firm, where he maintained the extended FORTRAN compiler. Steve says he enjoys the Oregon mountains and trees and plans to explore his new state through photography, skiing, and hiking.

As Manager of Customer Service, **Claire Lematta** helps customers with all phases of the ordering process and handles software support sales. She maintains the field test and software licensing agreements and can answer non-technical questions about them. Claire enjoys our varied customer base and is especially qualified to help overseas customers. She has lived all over the world, speaks Spanish, and has a degree in international studies. Claire is, however, a native Oregonian who enjoys cross-country skiing and backpacking as well as participating in the Oregon chapter of the World Affairs Council.

Mary Erichsen, OEM Manager, has been a senior sales representative at Wang and was a large account manager at Victor Technologies. At Oregon Software, she provides sales tools, workshops, and competitive information for existing and potential OEM customers. She recently negotiated agreements between Oregon Software and some top corporations in the industry. Mary travels throughout the United States to call on customers and to attend trade shows. Recently she broke her ankle while running in Boston, but she's still running. Be on the lookout for Mary at trade shows.

Tom Hanrahan joined Oregon Software in February as a technical writer. After obtaining his M.S. in engineering at U.C.L.A., he worked as a free-lance technical writer for three years. Now he is producing new manuals and updating old ones. Tom is another runner; he competes on weekends, if he's not cross-country skiing.

Hanh Hoang, our new data entry operator, puts ordering information from customers into our files. Hanh has her certificate of data entry operation from Portland Community College. When she's not working, Hanh enjoys dancing.

Gerry O'Scannlain is responsible for Oregon Software's advertising campaigns. She recently produced our handsome SourceTools package. Her next major campaign will be for Pascal–2. She brings extensive experience in marketing, graphics, and fund raising to her new job. Customers will be seeing Gerry's work at trade show presentations and as eye-catching, high quality ads in trade journals.

Louise Waitt recently joined the writing staff as an apprentice technical writer. In addition to her B.A. from Oregon State University, she earned a certificate in computer science at Denver University and worked in Colorado as a programmer. When her training is finished, Louise will be producing and maintaining OSI's technical manuals.

Dirk Eide began as a software engineer this February, after four years of experience in software and systems engineering as well as graduate coursework in math. Dirk is maintaining the RT-11, RSX, PDP-11, and UNIX systems for Oregon Software. His avocation is riding race horses at Portland Meadows Race Track.

Mark Paulin also joined our programming staff this February. He is modifying Pascal–2 compilers to run under UNIX on the 68000 series. Mark comes to Oregon Software after receiving his B.A. in math from Reed College and spending three years at Tektronix, where he designed and maintained software products.

Continued on Page 23

Pascal NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

David Spencer, editor
Ann Littlewood, Thomas E. Hanrahan, writers
Jennifer Mulder, production assistant

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Copyright © 1984 by Oregon Software, Inc.
ALL RIGHTS RESERVED.

RSTS, RSX, RT-11, PDP-11, VAX/VMS, and IAS are trademarks of Digital Equipment Corp. UNIX is a trademark of Western Electric. Pascal–1, Pascal–2, SourceTools and Pascal Newsletter are trademarks of Oregon Software.

Printed in USA

The Log: Pascal-1 and Pascal-2

Oregon Software's previous Pascal Newsletter described significant changes in Pascal-2 Version 2.1B. This log details bugs fixed in Version 2.1C, now available.

Pascal-2

Version 2.1C corrects these problems for PDP-11s with DEC operating systems.

ORIGINed variables misplaced

Addresses allocated to variables through the **origin** declaration were sometimes improperly assigned and led to incorrect code generation.

Set operation errors

The inclusion operator, **in**, generated bad instructions when certain elements were not recognized as belonging to a set.

Long comparison breakdowns

Particularly long, involved comparisons sometimes created an **undeleted temps** error and resulted in compiler shutdown.

'Exp' and 'Sqrt' problem

Under double precision, an inadequate range-checking routine for arguments sometimes caused **exp()** to return an incorrect value. Double precision **sqrt()** produced the wrong result when its argument was 0.

'Rad50' routines didn't work

Rad50 routines omitted the letter 'A' from the first and fourth positions of file names they were required to print. For example, **Rad50** would print the file name "HDRA.PAS" as "HDR.PAS."

String package miscues

Several logical and typographical errors in the string package have been corrected. In addition, two procedure names, **AssChar** and **DelString**, have been shortened to allow the entire string package to be placed in an external library.

Switching errors in /DEBUG option

Attempts to use the **/debug** option on external modules produced a spurious, **conflicting switches** error. With version 2.1C, you can debug external modules.

Changes to RSX

Version 2.1C for RSX corrected these problems:

The error in 'noioerror'

The **noioerror** procedure generated an incorrect value when reporting the status of errors involving the **put()** operation for random access files.

/LUN:n switch sensitivity

The **/lun:***n* switch, which is used to give a file a logical unit number other than the default assigned by Pascal-2, has been adjusted so that it is no longer sensitive to the order in which switches are applied.

'Forini' incompatibility

The FORTRAN initialization routine, **forini**, failed to work properly on VAX in RSX compatibility mode.

Oversized buffer files incompatible with SYSLIB

The support library no longer allocates large buffers when files containing oversized records (greater than 512 bytes) are opened. Previously, users were forced to use ANSLIB instead of SYSLIB. Pascal now allocates 512 byte buffers for all disk files unless the **/buff:***nnnn* switch is used. (Users taking advantage of the **/buff:***nnnn* switch must remember, however, that creating record sizes greater than 512 bytes again forces the use of ANSLIB.)

Changes to RT-11

Version 2.1C for RT-11 corrects these problems:

SIM errors

The simulation mode switch (SIM) for floating point arithmetic incorrectly handled **text** files and made the debugger inoperable.

'Rename' lacked a default

Rename now uses any field of the old file name that the new file does not specify as a default. Other, intermittent errors in **Rename** were also corrected.

Stack checking errors

Stack checking errors associated with completion routines written in Pascal have been eliminated.

Changes to UNIX

Version 2.1D corrects these problems for UNIX/68000 systems.

Code generation corrected

The compiler no longer generates incorrect code for operations with sets or for accessing structured constants fields of packed records.

Internals error corrected

Certain complex expressions no longer cause the compiler to generate the internals error.

Version 2.0N corrects these problems for UNIX/PDP-11systems.

PDP-11/23 problems fixed

The compiler now runs on the PDP-11/23; however, some features of the Debugger are still not available.

Source Tools

Version 1.0C corrects these problems.

'Newsrc' no longer prompts unnecessarily

Newsrc does not prompt for information that you've already supplied on command lines such as:

```
newsrc/input=test.pas/release=1/key=(descr="base
1[illegible]") mymodule
```

In previous releases, **newsrc** prompted for the module name **mymodule**.

'Make' functions corrected

The command

```
make test/debug=foo
```

now generates FOO.DBG instead of building the file MAKEFILE.DBG.

The command

```
make test/hier
```

no longer dies with a run-time error.

Changes to RSTS/E

Version 2.1C for RSTS corrects these problems:

Real numbers misread

When consecutive real numbers were to be read, the first character in the series was missed.

Run-time errors not distinguished

Run-time error checking routines now provide negative error numbers for operating system errors and positive ones for error messages from the Pascal support libarary.

Sayerr error

Sayerr now does not print, `RSTS error...` before the body of the error statement.

Known bugs

Our last newsletter promised customers a statement of company policy on known bugs. Future sets of release notes will contain a list of unfixed bugs and workarounds, if known. We will also print this information in the newsletter, starting with the next issue. We hope these procedures will spare customers both the annoyance of discovering a bug and the unnecessary labor of reporting an already known bug.

Errors, additions to manuals

The RSTS Pascal 2.1 manual, page 2-59, omits a command line essential for calling FORTRAN from Pascal. After the command line:

```
LINK FTEST=FTEST,INNAM,P:PASCAL,$FORLIB
```

as listed in the manual, add:

`PRTS FTEST n` (where n is some memory size 2 to 28).

This command allocates enough memory for the program to run and associates the Pascal runtime system with the .SAV file. Otherwise the program tries to run with the RT-11 runtime system, the "default" for FORTRAN and for programs that are built on RSTS using the `LINK` command. Without the second command line, the program will get an "M-Trap to 4" error immediately and die.

Case Study

Programming a process-control system

by Kurt W. Papke

Mr. Kurt W. Papke was formerly the Engineering Manager at Spectronix, a small systems house that writes software for real-time process control systems for handling bulk materials, primarily for the grain industry. In this article, he describes the trials and tribulations of programming one of Spectronix's early projects, then relates the scope of an even larger current project.

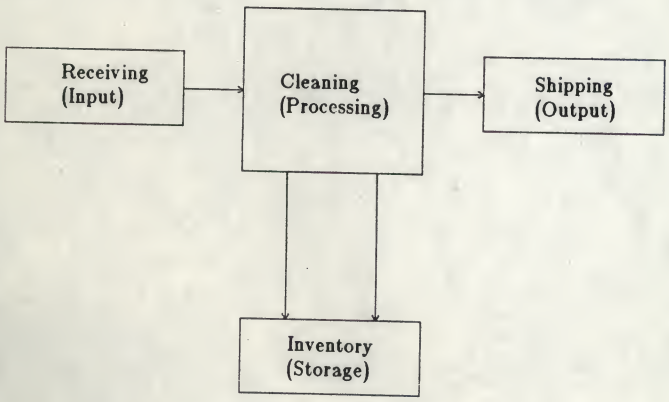
Real-time industrial process control systems, with their huge databases, mulitple concurrent processes, complex control algorithms and immense quantities of I/O points, have an innate tendancy to tax hardware, software, and programmers to their limits.

The Ups and Downs of a Grain Elevator

The modern grain elevator is a complex system that handles grain much like computers handle data. In computer terms, an elevator is a large I/O buffer. Terminal elevators "buffer" domestic grain for export overseas to our friends in Russia, China, etc. Commodities are input from rail cars, barges, or trucks, then held in storage bins, and finally output to ships.

The mechanism for implementing this big I/O buffer is a mechanical engineer's nightmare. A typical grain elevator consists of conveyors, valves, modulated gates, elevating legs, and electronic scales, not to mention several hundred bins. The investment of several hundred million dollars of capital investment represented by this equipment easily justifies the expense of a computer control system.

Figure 1. Elevator as I/O Buffer



Conceptually, a grain elevator looks like a large directed graph or "digraph." The edges of the digraph are made up of the conveyors and elevating machinery, and its nodes are the holding bins and switching equipment. It is the job of the computer system to control, monitor and report on the movement of the grain through this digraph. Consider the flow diagram of a typical terminal elevator shown in Figure 2.

The control system directs the grain through the elevator by controlling solenoid or hydraulic switching mechanisms at each "fork in the road." It must be careful to avoid collisions of grains from different sources. Extensive exception handling has to be built into the logic to cope with situations like bins filling and equipment failure.

Typical Process-Control Architecture

A typical control system for this industry uses a classic three-level hierarchy (shown in Figure 3).

The top level preforms supervisory functions for the entire plant, making strategic decisions and serving a data base function. Typical machines at this level are of the VAX class, due to the data base orientation of their function.

The second level of the hierarchy controls a specific area of the physical plant such as a rail-receiving stream. The processor at this level makes local tactical decisions, usually based upon a massive amount of input. Typical machines at this level are PDP/11-44s, chosen for their high I/O handling capability and memory capacity.

At the peon level lives the Programmable Logic Controller, a strange beast manufactured by the likes of Modicon, Allen-Bradely and Texas Instuments. They are programmed in the relay-ladder logic and are particularly well-suited to interfacing to the world of motor starters, rotary encoders, and limit switches.

Why Pascal?

The vast majority of process-control applications are coded in a combination of assembler and FORTRAN. Assembler is used for "efficiency" (an illusory metric, of concern mostly to the person who purchased too little memory for the CPU). FORTRAN is used because process-control systems are designed by engineers, who only learned FORTRAN in college, and who think that the only construct required in a programming language is an arithmetic IF.

Figure 2. Example of an Elevator Digraph

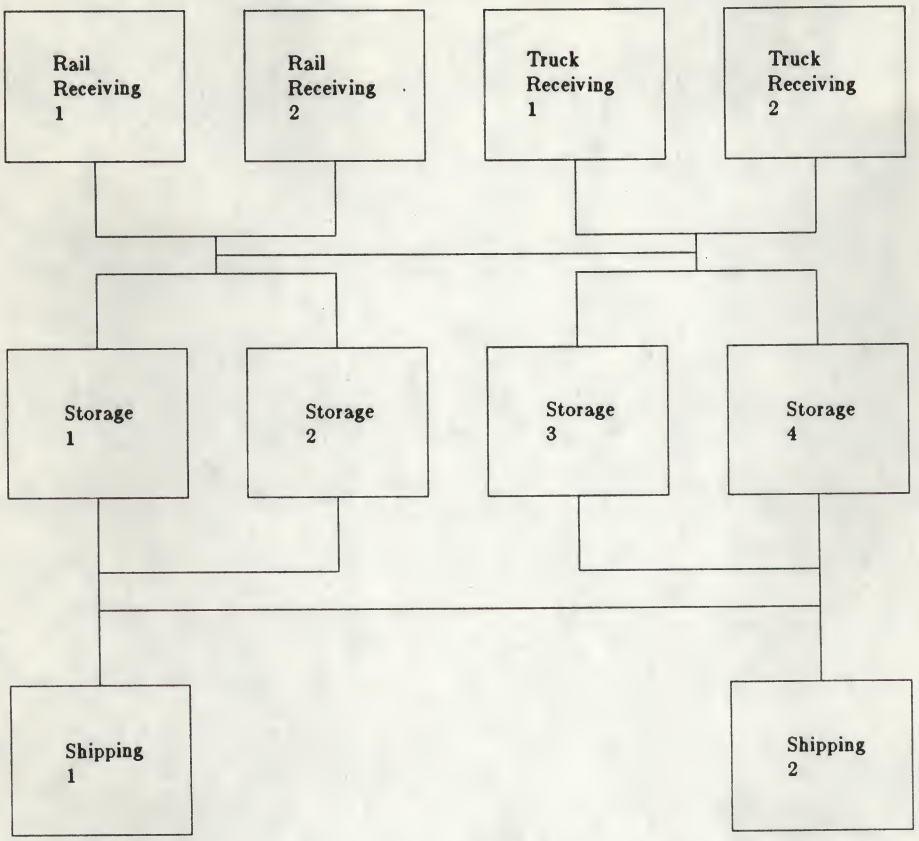
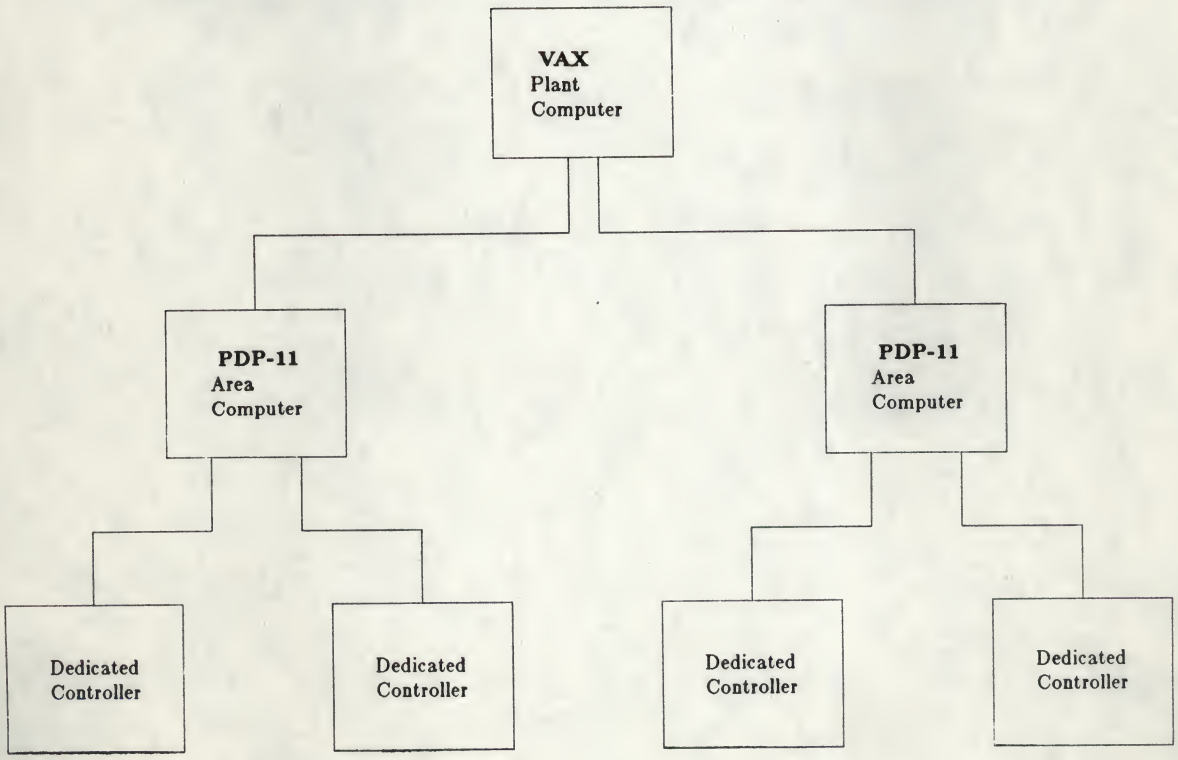


Figure 3. Classical Control System Design



We chose to use Pascal because of the extremely complex nature of both the algorithms and data structures we must deal with, such as digraphs, shortest path, linear programming, etc. A typical shared common region, which is the memory-resident data base that drives the control system, is made up of arrays of records of arrays of records of....

Such complex structures are awkward to handle in assembler or FORTRAN. Another factor is the Mongolian Horde of Pascal programmers graduating from the universities; it is becoming far easier to find Pascal programmers than practitioners of any other language.

So far the decision seems to have been a wise one. The language has some major deficiencies, but considering the alternatives, Pascal appears to be the least evil. Other languages considered as a replacement were Modula-2 and Ada.

An Early Effort

The first large system we designed was for a Gulf Port export elevator. The system was intended to replace the traditional control panel consisting of lights and switches with color graphic displays and a supervisory minicomputer.

The computers were a bit undersized: MC6809 processors for the tactical level and a single PDP-11/44 for the supervisory and data base machine. The 11/44 was equipped with 1 megabyte of RAM, dual RK07 drives and 32 serial ports (gasp!). The system software consisted of RSX-11M V4.0, RMS-11K, Datatrieve, FMS-11 and Oregon Software's Pascal-1. We considered converting to Pascal-2, but abandoned the idea because of the enormous effort such conversion required.

The applications software consisted of roughly 75 Pascal programs in 40 directories making up 50,000 lines of code (comments included). The design and coding effort required 7 man-years in a 16 month calendar period using 5 programmers. If you work out the numbers, this averages out to 30 debugged and documented lines of code per day per programmer, versus the supposed industry average of 10 lines. In terms of actual dollars, this software costs roughly $10 per line of debugged code.

In order to complete the project, we had to deal with a number of issues concerning the use of Pascal-1.

- Source code sharing
- Calling RSX directives
- Writing asynchronous system trap (AST) routines in Pascal-1
- Use with FMS-11
- Use with RMS-11K
- Use with multiple system resident common blocks
- Symbol table space shortages

Our problems were caused by: difficulties inherent in Pascal itself, the FORTRAN orientation of RSX-11, and the constraints imposed by Pascal-1, many of which have been eliminated in Pascal-2.

Source Code Sharing

In any large software system, there is a need to share common source code, notably utility routines and data type definitions. Unlike Pascal-2, the Pascal-1 compiler has no dynamic `%include` directive. We had a choice of two methods for working around this problem: code up a utility program to do the inclusion or use a DEC standard utility in a non-standard manner.

We decided to use SLP because it was a very reliable, DEC-supported utility. We had enough applications software to worry about without the additional concern of utility programs. The SLP input file for a program looked like:

```
PROG.PAS/-AU=
PROGRAM Prog;
@filename1.pas
@filename2.pas
BEGIN
END.
/
```

SLP processes this file to produce PROG.PAS, bringing in all files indicated by the @ signs at up to three levels of indirection. This technique has worked out quite well and we continue to use it with other utilities that do not support dynamic inclusion, such as Oregon Software's PROSE text formatter.

Calling RSX directives

In any real-time system, programs must access the operating system directives. DEC supports only MACRO-11 and FORTRAN interfaces to the RSX-11M directives. The FORTRAN callable routines are difficult to use from Pascal because they do expect the FORTRAN run-time system to be present. They are also somewhat restrictive and do not take advantage of any of Pascal's data structuring capability.

We created an object library of Pascal-callable RSX directives, some of them designed to exploit Pascal structures. For example, we changed the "wait for logical OR of event flags" (WTLO$) to a "Wait for any in a set of event flags,"

in order to hide the use of the event flag mask. In this form the directives turned out to be more readily understood by Pascal programmers. An example call might be:

```
Const MaxEfn = 64;

Type EfnRange = 1..MaxEfn;
     EfnSet = Set of EfnRange;

Procedure WaitForSet (Efns : EfnSet); External;
```

Writing AST Routines in Pascal-1

Due to the large amount of send/receive data and intertask communication involved, we needed a method to write receive data ASTs in Pascal. So, we wrote a short MACRO routine that took as a parameter the name of the procedure to be called for AST servicing. An "envelope" routine was specified in the SRDA$ call, which invoked the MACRO procedure, set up R5 for the heap pointer, and executed the AST exit (ASTX$) upon completion.

This worked out quite well, allowing the routine to be completely coded in Pascal. The service procedure could communicate with the main code via event flags or global variables. The code for the "Specify Receive Data AST" routine was:

```
Procedure RsxSrda (Procedure AstRtn);

    {---(MACRO Envelope Routine)---}

{$C        .IDENT "820311"     ; Creation date
$$R5$$:    .Word 0             ; Heap address
           .Mcall Astx$
Astx$:     Mov    R0, -(Sp)    ; Save Register
           Mov    R1, -(Sp)
           Mov    R2, -(Sp)
           Mov    R3, -(Sp)
           Mov    R4, -(Sp)
           Mov    R5, -(Sp)
           Mov    $$R5$$, R5   ; Get heap pointer
           Jsr    Pc, @(pc)+   ; Call the procedure
AstAdx     .Word 0             ; Routine Address
           Mov    (Sp)+, R5    ; Restore registers
           Mov    (Sp)+, R4
           Mov    (Sp)+, R3
           Mov    (Sp)+, R2
           Mov    (Sp)+, R1
           Mov    (Sp)+, R0
           Astx$s              ; Exit from AST
}

    {---(Actual Procedure Body)---}

Begin {RsxSrda}
    {$C
        .Mcall Srda$s
                               ; Remember heap pointer
        Mov    R5, $$R5$
                               ; Store routine address
        Mov    2(sp), AstAdx
        Srda$s AstEnt          ; Specify entry point
    }
End; {RsxSrda}
```

Use with FMS-11

Using Pascal-1 with FMS-11 is one of the most frustrating experiences imaginable. The problem has two facets: Pascal-1 has no variable length string facility, and FMS requires ASCIZ strings (terminated with a null character). The frustration level is such that, after writing every FMS task, we'd swear up and down never to use Pascal again!

There is no clean solution to this problem. If you use the Pascal-1 string package, you still must hassle with string constants, which necessarily occur everywhere in an FMS task, for field names, form names, etc.

The best we could do in Pascal-1 was to define the FMS procedure parameters to be single characters passed by address (**var**), and then fool the compiler's type checking by passing only the first character of the string. Unfortunately, the Pascal-2 compiler negates this "work-around" because string constants are **packed** arrays and an element of a **packed** array cannot be passed as a **var** parameter. Foiled again!!

The introduction of Version 2.1 of Pascal with its conformant array parameters means a solution is in sight. We are in the process now of writing a package that uses conformant arrays to pass the string parameters to FMS-11.

Use with RMS-11K

The use of Pascal-1 with RMS-11K is by now a relatively well understood procedure (see the "Information Exchange" in Pascal Newsletter Number 4). The remaining difficulties consist primarily of keeping the Pascal record definitions in synch with the Datatrieve record definitions as the fields in the files change. We've attempted to automate this process by writing a translator for Datatrieve-to-Pascal record definitions.

Multiple System Resident Common Blocks

The use of common blocks with Pascal-1 (or Pascal-2) is a simple matter: the `origin` statement can be used or an external MACRO-11 procedure can be written to set a pointer variable to the start of the common region.

One caveat however: the use of pointer variables within a system common is dangerous—care must be taken that every program maps the commons to the same spot in their virtual address space. Programs that map a common region containing pointers to different address spaces end up with odd address traps, memory protect violations, etc. Normally, these problems only occur when multiple commons are in use, and not every program specifies the APRs to be used in the task build command file.

Symbol Table Space Shortages

Due to the relatively large size of the program (when type definitions and service routines were included), running out of compiler symbol table space was a chronic problem. We never found a solution for this, other than upgrading to Pascal-2, which does not limit symbol table size to the available heap space.

The Current Effort

In the continuing saga of Spectronix versus process control, the next chapter is a large, state-of-the-art facility. We intend to provide a high-speed, fault tolerant system with the latest in man-machine interfaces.

Figure 4 shows the modern LAN-based architecture of the new system, which contains four VAX-11/750s, four PDP-11/44s, and nine Allen-Bradley PLC-3s. The DEC machines are connected via a PCL-11 (pronounced pickle) and the 11/44s talk to the PLC-3s over the A-B Data Highway. RM81 disk drives give the system more than 3 gigabytes of disk storage. All the disks are dual-ported to allow access in the event of a machine failure. Operators communicate with the system through a voice recognition system (VOTAN) and alarms are annunciated with voice output. Process status is displayed on twelve VS-11 color graphics screens, four of which are wall-projected to ten-foot diagonal displays.

The challenge is to program all those CPUs with what we estimate will be 100K lines of Pascal and to solve these problems:

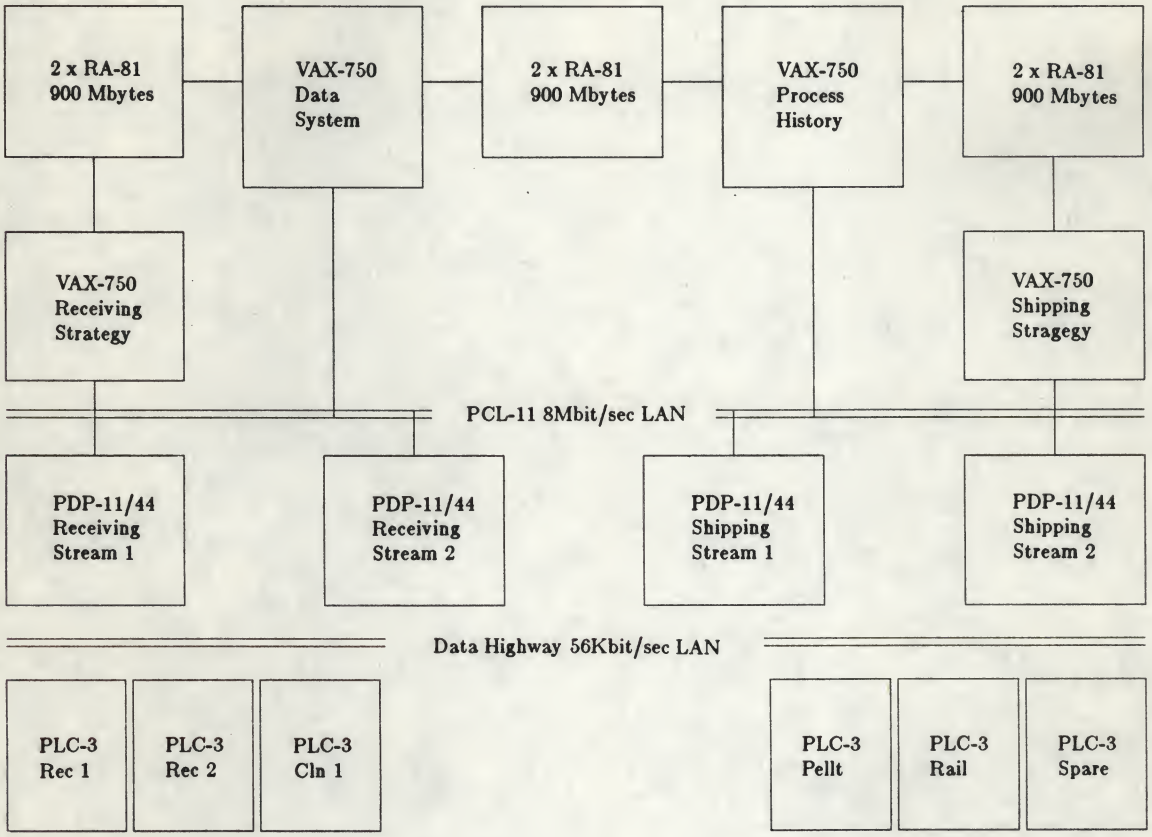
- Two different CPUs requiring two different Pascal compilers. It would have been nice to use an Oregon Softwa compiler on the VAX but....
- Utility (FMS,RSX,RMS) libraries upgraded to run under Pascal-2 on the 11/44s.
- Large common blocks (>32K words) on the 11/44s implemented with PLAS directives to make use semi transparent from Pascal.
- DECnet everywhere.
- Real-time graphics including process animation (we are going to show the rats chewing grain in real-time)
- Voice command and response.
- Fail-safe design—if one machine goes down anothe must take over screwing things up where the failed machine left off (more or less).

All this has to be done sometime in 1984. Wish us luck Lord knows we are going to need it!!

Editor's Note:

Since Mr. Papke wrote this article, a VMS version of Pascal–2 is much closer to completion. Pascal–1 has a limited variable-length string facility; its structure is not generally compatible with other languages.

Figure 4. Modern LAN Based Architecture



OPUS Communiqué

Oregon Pascal Users Society
by Bruce Williams, Coordinator

The OPUS conversion from RSX-11M to RSX-11M/+ will be finished soon, followed shortly by the official opening of the OPUS library. Get ready to submit your routines!

Some members have requested the DCL/MCR Command processor routines. This will be done as soon as the library is up and running.

The membership list is in the mail. Space does not allow printing it in this newsletter; it will appear in the next issue.

Thanks for your time and patience!

Micro/RSX available, Pascal-2 compatible

Digital reports that Micro/RSX is now available on the Micro/PDP-11. The Micro/PDP-11 uses the PDP-11/23-PLUS central processor and executes the full set of PDP-11 instructions. A customer tells us that Pascal-2 runs on this system, but you must have the "Advanced Programming Package" and select "NOANSLIB" instead of "SYSLIB" as your system library.

Continued from Page 14

Oregon Software's new Distributor Sales Representative is **Linda Lausmann**. She is the main liason with international distributors, processing sales orders, solving problems, and answering correspondence. In addition to her B.A. in business administration, Linda is experienced in both accounting and sales. **Marilyn Crossgrove**, her predecessor, is moving to England.

Oregon Software's staff is not static. As a company, we try to facilitate the personal and professional growth of each member by encouraging many horizontal and vertical career paths. Since our last newsletter, many have taken new paths.

Collins Hemingway was the original technical writer at OSI and over a period of two years, he established the Technical Publications group. Collins is now the public relations liaison between the company and trade journals, placing technical articles and coordinating informational tours.

Terry Juve has moved from distributor sales to the technical group. As a programmer trainee, she's processing trouble reports and testing software.

Diane Likens has been promoted to Customer Service Representative. She controls inventory for Oregon Software headquarters and the European Warehouse. In addition, she handles special manual orders and special projects, such as improving our record keeping procedures.

Betsy Slonaker, OSI's word processing operator, is now Production Assistant for Technical Publications. She'll be getting manuals and newsletters ready for printing and training in computer operations and the TEX system.

Laura Ronck has been promoted to Programmer. She performs validation tests for new Pascal–2 releases for DEC and Motorola systems; she also sets up release mechanisms.

Senior Software Engineer **Steve Poulsen** now serves as technical consultant to Sales/Marketing. He provides answers to customers' questions about the software and has been working closely with Mary Erichsen on OEM sales. Steve is also a frequent contributor to this newsletter.

As Vice President of Software Development, **Bob Phillips** is hiring new staff and reorganizing OSI's development and support group to prepare for future expansion.

As Purchasing Agent, **Katie Wilding** negotiates with all vendors and handles travel arrangements. She still fills in as receptionist occasionally.

Sandy Profeta has been promoted to Systems Manager and has the full responsibility for maintaining OSI's three DEC systems. She also negotiates purchases of magnetic media for product distribution and new hardware for OSI's computer system.